Supplemental Material: QuadStream: A Quad-Based Scene Streaming Architecture for Novel Viewpoint Reconstruction

JOZEF HLADKY, Max Planck Institute for Informatics, Germany and NVIDIA, Germany
MICHAEL STENGEL, NVIDIA, USA
NICHOLAS VINING, University of British Columbia, Canada and NVIDIA, Canada
BERNHARD KERBL, TU Wien, Austria
HANS-PETER SEIDEL, Max Planck Institute for Informatics, Germany
MARKUS STEINBERGER, Graz University of Technology, Austria

CCS Concepts: • **Computing methodologies** → **Rendering**; **Texturing**; **Virtual reality**; Image-based rendering.

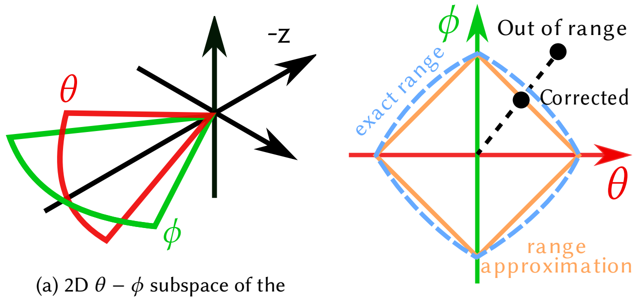
Additional Key Words and Phrases: texture-space shading, object space shading, shading atlas, streaming, temporal coherence, virtual reality

ACM Reference Format:
Jozef Hladky, Michael Stengel, Nicholas Vining, Bernhard Kerbl, Hans-Peter Seidel, and Markus Steinberger. 2022. Supplemental Material: QuadStream: A Quad-Based Scene Streaming Architecture for Novel Viewpoint Reconstruction. *ACM Trans. Graph.* 41, 6, Article 233 (December 2022), 5 pages. https://doi.org/10.1145/3550454.3555524

1 ORIENTATION CORRECTION

As the Quad Frustum Space (QFS) is centered around $(0, 0, -1)$, our rotation range can only support front-facing orientations. Thus, the 3D orientation vectors of quad proxy planes can be transformed into a 2D $\theta - \phi$ subspace of the spherical coordinate system (Fig. 1(a)). The resulting orientation range is a symmetrical shape in the $\theta - \phi$ space that we determine through sampling (Fig. 1(b)). This shape is very close to a diamond with only slightly convex curves, which allows for an efficient conservative approximation using straight lines. If we detect an out-of-range orientation—a point outside of the supported orientation range shape—we obtain the minimally intrusive orientation correction by taking the line from the orientation towards the origin of the $\theta - \phi$ space and intersecting it with the shape describing the supported rotation ranges. In our experiments, the *overcorrection* of the rotations is very small in practice. As every QFS centers the quad frustum around the Z axis, the ranges are invariant to the screenspace location of the quad proxy and therefore depend only on the quad proxy's size.

Authors' addresses: Jozef Hladky, Max Planck Institute for Informatics, Germany and NVIDIA, Germany, jhladky@mpi-inf.mpg.de; Michael Stengel, NVIDIA, USA, mstengel@nvidia.com; Nicholas Vining, University of British Columbia, Canada and NVIDIA, Canada, nvining@cs.ubc.ca; Bernhard Kerbl, TU Wien, Austria, kerbl@cg.tuwien.ac.at; Hans-Peter Seidel, Max Planck Institute for Informatics, Germany, hpseidel@mpi-sb.mpg.de; Markus Steinberger, Graz University of Technology, Austria, steinberger@icg.tugraz.at.



(a) 2D $\theta - \phi$ subspace of the spherical coordinate system

(b) Orientation correction

Fig. 1. The orientation of quad proxy planes is transformed into a 2D subspace of the spherical coordinate system. This subspace enables to easily correct extreme orientations.

2 ADDITIONAL FIGURES

Figure 2 and 3 show extended figures of the main evaluation figures from the paper.

3 IMPLEMENTATION

In this section we discuss implementation details of the QuadStream pipeline, depicted in Fig. 4.

In our implementation, we sample the view cell for a set of views, from which we construct G-Buffers; the G-Buffers are used to construct quad proxies for the given projection, potentially splitting a quad proxy into multiple planes to capture disjoint surfaces. We internally arrange the quad proxies into a server-only data structure, called the quad map, which we process to merge neighboring quads and then bin the merged results according to size. After processing the initial view, each following proxy view works adds only yet-uncaptured surfels to the quad map. Each stage of the quad proxy processing pipeline is run in compute shaders. When processing of all proxy views is completed, we traverse the quad map in parallel and arrange the binned surfels into an atlas texture. The atlas texture is then compressed and streamed over the network to the client.

3.1 Construction and Splitting

Construction and splitting of quad proxies from the G-Buffer is performed in a compute stage, which operates on rectangular blocks of texels of size b_x, b_y in the G-Buffer. While our method works in parallel, we outline a serial algorith for clarity (Alg. 1). We use a single warp per texel block of size $b_x = b_y = 2$ plus 1 texel boundary,

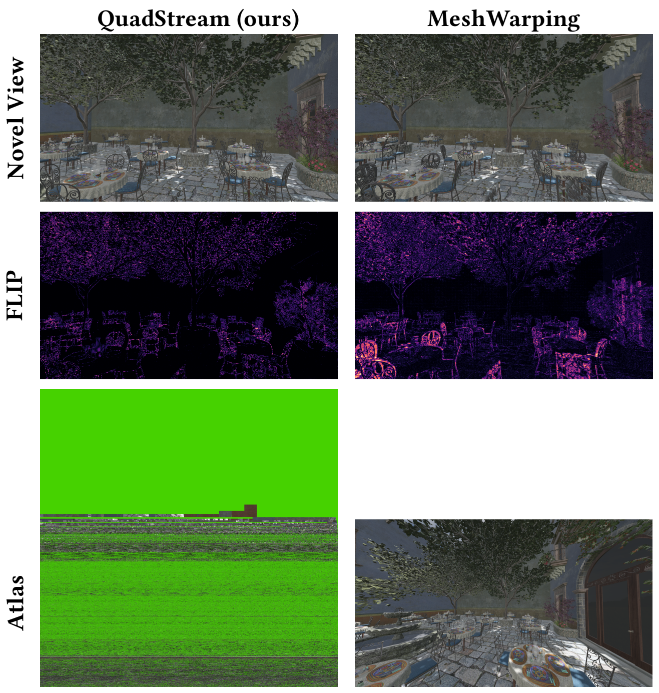


Fig. 2. A comparison of the novel view renderings for San Miguel.

i.e. 4×4 texels. The block texels, together with the boundary, form the *surfel pool* of the block.

After loading the surfel pool from the G-Buffer, we construct a local shared-memory fixed-size array of size $(b_x + 2) \times (b_y + 2)$ and populate it with candidate quad planes for voting (48 bits per element, line 2). Surfels are assigned to candidate quad-planes by a two-stage voting and matching process. First, we construct a set of candidate quad planes, one from each surfel (line 4). Each surfel then votes on the fitness of each candidate plane, iterating over shared memory (line 8). During the voting process, each surfel computes its projected depth offset to the candidate plane; if the offset is below the threshold $\delta_{\max}$, the energy of the quad candidate is increased. After all surfels vote, we synchronize the threads and perform in-block parallel sorting of the candidate quad plane array based on the evaluated fitnesses. At this point, the quad fitness expresses how well the given quad approximates the geometry of all surfels within each processing block.

In the second stage, each surfel traverses the sorted candidate quad array looking for the first match onto which it projects within $\delta_{\max}$. Upon finding such a candidate, the surfel marks its G-Buffer coordinates into the internal memory of the quad candidate (line 13). Since the quad candidates are sorted according to their fitness, the candidates which best approximate the whole surface are prioritized. The output of this stage is 1 to $(b_x + 2) \times (b_y + 2)$ planes stored in a quad map—a 3D buffer which follows the spatial layout of the G-Buffer. For a G-Buffer with resolution $(g_x \times g_y)$, the quad map dimensions are $(\frac{g_x}{b_x}, \frac{g_y}{b_y}, (b_x + 2) \times (b_y + 2))$. The quad map is used only by the server; it is not streamed to the client, nor used for novel view extrapolation.

ALGORITHM 1: Quad Construction

```
1  load ($SurfelPool$)
2  shared $candidateQuads[]$
3  for $s \leftarrow SurfelPool$ do
4      $p \leftarrow s$.constructQuadPlane()
5      $candidateQuads$.emplace($p$)
   end
6  for $s \leftarrow SurfelPool$ do
7      for $g \leftarrow candidateQuads$ do
8          $s$.voteFitness($g$, $threshold$)
       end
   end
9  $candidateQuads$.sort()
10 for $s \leftarrow SurfelPool$ do
11     for $g \leftarrow candidateQuads$ do
12         if $s.distance(g) \leq threshold$ then
13             $g$.assignSurfel($s$)
           end
       end
   end
14 global $finalQuads[]$
15 for $g \leftarrow candidateQuads$ do
16     if not $s.surfels.empty()$ then
17         $finalQuads$.emplace($g$)
       end
   end
```

3.2 Merging & Flattening

Groups of neighbouring quads are identified and merged by processing the 3D quad map on the GPU. $\lfloor log(min(g_x, g_y)) \rfloor + 1$ consecutive compute kernels are iteratively launched which operate in parallel. Each kernel launch l loads 4 neighboring quad proxies with a stride of 2^l. We split the warps into groups of 4 threads, where each thread loads all final quad planes for the four neighboring quads. We unproject each quad plane back into viewspace, load them into shared memory, and perform comparison of the quad plane parameters. If the computed parameter difference is under a fixed plane similarity threshold t_{Sim} (we use 0.1), we use the 4 planes to compute the average plane (by averaging the plane parameters) and mark it as ready for processing in the next kernel launch level. If merging of four neighbors is not possible or we reached the last kernel launch $l = \lfloor log(min(g_x, g_y)) \rfloor + 1$ we then transform the planes back into Quad View Space for the merged quad proxy, retrieve each surfel of the merged sub-quads, and recompute the surfel depth offsets projected onto the new quad plane. At this point we have successfully determined the final size of the quad proxy and we increase an atomic counter for the bin corresponding to its size in order to mark the position of the surfels in the final atlas texture. We again provide a serial pseudo algorithm in Alg. 2.

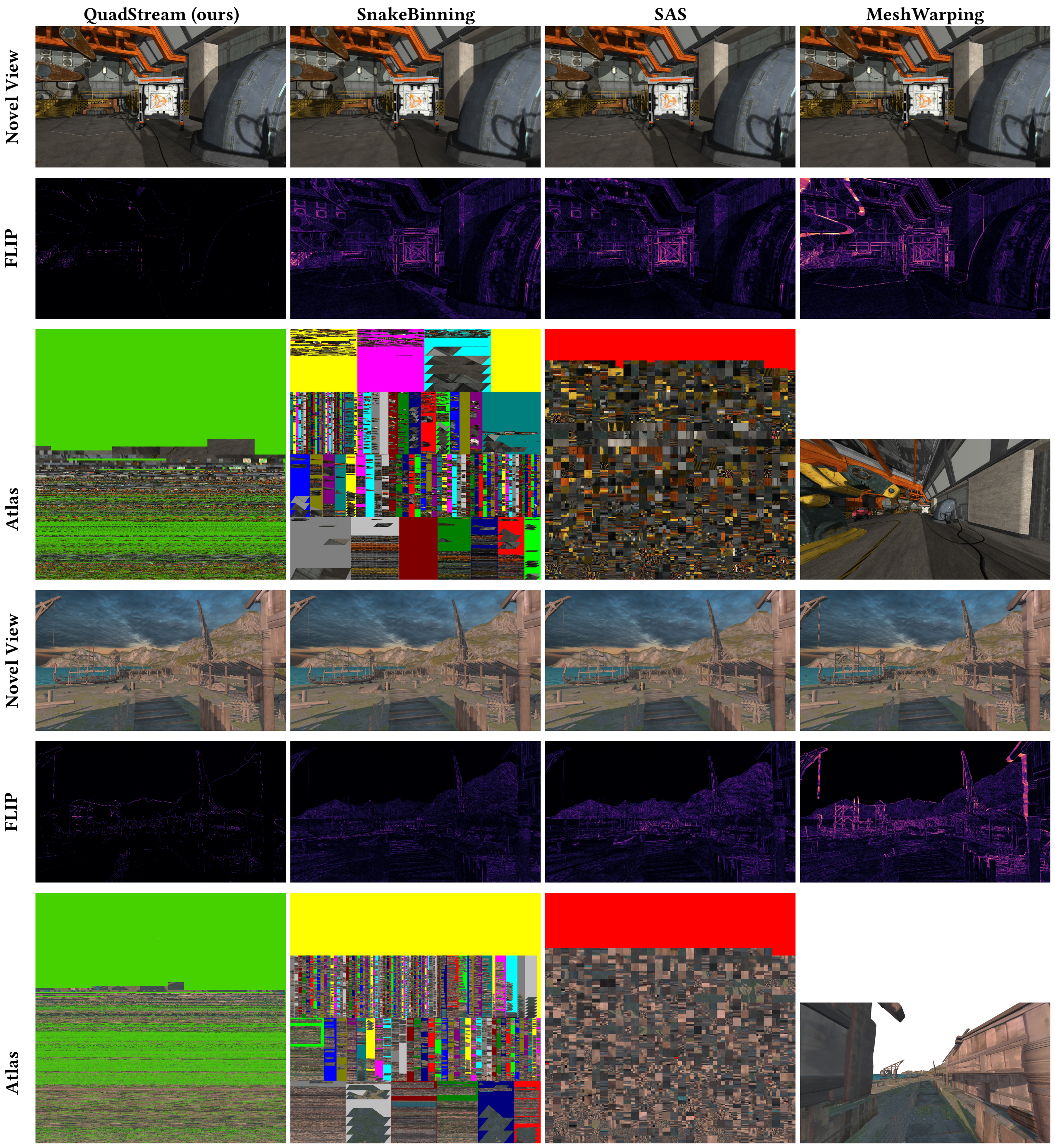


Fig. 3. A comparison of the novel view renderings for all tested methods. FLIP [Andersson et al. 2020] perceptual error was used, measured against ground truth for the novel view. The atlases used by each method are rendered at equivalent resolution. Note that we both require significantly fewer samples than previous methods and make more efficient use of texture space in the packed atlas, while also improving on FLIP error vs ground truth for these novel views. Please zoom in for details.

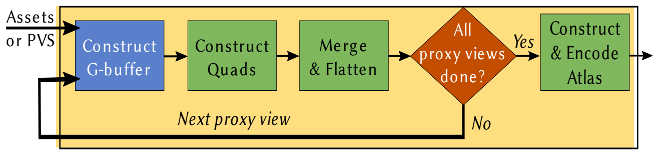


Fig. 4. QuadStream server pipeline: We iterate over a set of sampled views in a view cell; for each view we render a G-Buffer, construct quad proxies following the rectangular grid of the view projection plane, merge suitable neighboring quad proxies, and potentially flatten them. After obtaining a set of quad proxies for all views in the view cell, we construct an atlas containing both the geometry and shading data and stream it to the client device. The client uses this package to render novel views and thus perform framerate upsampling.

ALGORITHM 2: Quad Merging

```
1  for $(x, y) \in (g_x/2^l, g_y/2^l)$ do
2      $Q_{\text{proxies}}[4] \leftarrow$loadNeighbours($x$,$y$,$l$)
3      for $Q_{bLeft} \in Q_{proxies}[0]$ do
4          for $Q_{bRight} \in Q_{proxies}[1]$ do
5              for $Q_{tLeft} \in Q_{proxies}[2]$ do
6                  for $Q_{tRight} \in Q_{proxies}[3]$ do
7                      if *similarEnough(simThreshold)* then
8                          merge($Q_{\text{bLeft}}, Q_{\text{bRight}}, Q_{\text{tLeft}}, Q_{\text{tRight}}$)
9                          $Q_{\text{proxies}}[0]$.remove($Q_{\text{bLeft}}$)
10                         $Q_{\text{proxies}}[1]$.remove($Q_{\text{bRight}}$)
11                         $Q_{\text{proxies}}[2]$.remove($Q_{\text{tLeft}}$)
12                         $Q_{\text{proxies}}[3]$.remove($Q_{\text{tRight}}$)
                       end
                   end
               end
           end
       end
13     for $i \leftarrow 0 : 3$ do
14         if not $Q_{proxies}[i]$.*empty()* then
15             for $p \leftarrow Q_{proxies}[i]$ do
16                 if *allSurfelsBelowDepth(p.surfels, dMaxOff)* then
17                     flattenQuadProxy($p$)
                   end
18                 $p$.positionInBin $\leftarrow$ nextFreeBinPosition($p$.size)
               end
           end
       end
   end
```

3.3 Adding Proxy Views

After computing the quad proxy set for the initial view, we augment it by constructing additional quad proxies from offset viewpoints which contribute only the disoccluded areas not yet captured in our quad proxy set (Fig. 4). For each additional proxy view sampled from the view cell, we first fill the depth buffer using the quad proxy set that we obtained so far. This is used as an early fragment rejection step for constructing a G-Buffer for the offset viewpoint. We then execute the Quad construction, split, merge and flatten step for the areas of the G-Buffer where new surfels were captured. As a last proxy view we use a low-resolution wide FOV projection encompassing the whole view cell to further reduce the possibility of disocclusion artifacts appearing in the novel views.

3.4 Atlas Construction & Encoding

After all proxy views are processed, we traverse the quad map in order to arrange the surfels into our raw atlas textures. To this end, we launch a compute shader that traverses the quad map and queries the atlas position of each quad within the bin. Since we work on a very small number of bins (10 bins, supporting a max quad size of 2^{10}, one bin per each power of two) and we have their occupancy, we determine the per-bin offsets into the shading atlas on the CPU. We ensure to keep a maximum overlap with memory allocations from the previous frame, *i.e.*, allow for padding to create more overlap. We then launch a first kernel to determine which allocations from the previous frames can be reused. To this end, we identify whether a quad proxy of same size has been generated at the same location inside the same offset view. If this is the case and the allocation still falls inside the atlas area assigned to the bin, we mark the allocation for reuse. In a second kernel launch, we traverse the quad map, allocate atlas locations for those quad proxies which do not reuse allocations, load the corresponding surfels for each quad and stitch them together in the final atlas texture. After obtaining the final raw textures, we compress them as described in Sec. 4.6 pack them for streaming to the client.

3.5 Client Rendering

The client rendering can be implemented in three ways: using tessellation shaders, fragment shaders (with ray-marching) or mesh shaders. Depending on the capabilities of the client GPU, different implementations are preferable. We at first describe the implementation for NVIDIA GPUs, followed by the discussion for current mobile GPUs. Although it is possible to process all quad proxies with only one approach, rendering flattened quad proxies using mesh or tessellation shaders is highly inefficient. We therefore process flattened quad proxies using a standard pipeline consisting of vertex, geometry and fragment shaders. We use mesh shaders only for the non-flattened quad proxies.

Using mesh shaders, we reconstruct the geometry for a single view directly from the received quad proxy buffer. We load a single quad proxy per task shader invocation (non-flattened) or per geometry shader invocation (flattened). For the case of a quad proxy that is a flattened quad (that is, it has no associated surfels), we unproject the quad plane back into world space using the $\mathbf{B}^{-}1$ matrix and matrices of projection $\mathbf{P}$ from its corresponding proxy view; intersect it with the associated QuadFrustum to spawn four corner vertices; and finally transform the corner vertices using the camera parameters of the novel view. For each vertex with 2D index v_x, v_y, we determine its normalized UV coordinates on the plane surface as $u_q = (v_x \frac{1}{w-1}) - \frac{1}{2} v_q = (v_y \frac{1}{h-1}) - \frac{1}{2}$. For a quad proxy centered around G-Buffer texel coordinates s_x, s_y and G-Buffer resolution r_x, r_y, we obtain the vertex global UV coordinates in the G-Buffer $u_g = u_q \times \frac{w}{r_x} + \frac{g_x}{r_x - 1}$, $v_g = v_q \times \frac{h}{r_y} + \frac{g_y}{r_y - 1}$. The view-ray for the given vertex in Quad View Space is then defined by two points: $R_A = \mathbf{BP}^{-}1 \begin{bmatrix} u_g & v_g & nearZ \end{bmatrix}$ and $R_B = \mathbf{BP}^{-}1 \begin{bmatrix} u_g & v_g & farZ \end{bmatrix}$. We obtain the intersection point between the view ray and quad plane $\mathbf{I} = \begin{bmatrix} i_x & i_y & i_z \end{bmatrix}$, project it to clip space $I_{clip} = \mathbf{P} \times \begin{bmatrix} I & 1 \end{bmatrix}$ and obtain its depth $D = \frac{I_{clip_Z}}{I_{clip_W}}$. The vertex then queries the atlas texture for its depth offset d and obtains its final depth $D_{final} = D+d$ in projection P. We then unproject this surfel back to world space, from Quad View Space, via the view space.

The Quest 2 is capable of achieving its 72 FPS target framerate at the suggested eye resolution (1440×1584) while rendering as many as 700k–1M triangles each frame. To unlock this performance, we enable dynamic fixed foveation rendering and further avoid tessellation/geometry shaders to enable compatibility with the `OVR_multiview` extension. Instead, we unpack and pre-tessellate received QuadStream buffers to yield necessary per-vertex attributes (position + UV) which we store in a vertex buffer. Doing so takes 19ms for Robot Lab and 25ms for San Miguel, however, note that these tasks must only run once for each received capture, and can be implemented as part of a background task for handling input stream updates. Finally, we disable costly order-independent blending between quads; instead, we pre-fill the color buffer by solidly rendering all quads marked as transparent and discarding associated depth buffer updates. For scenes with only opaque objects, doing so suffices to remove any visible gaps between quads.

Similar to other mobile GPU architectures, the XR2 chip of the Quest 2 bins triangles before performing tiled rendering. Due to the high number of triangles and the simple fragment shading, a significant amount of each frame ($23-28\%$) was spent in the binning stage in both scenes. However, we observed that the implicit spatially ordered layout of quads in the buffer and atlas already alleviates this burden: compared to an input with the same quads ordered randomly, binning completes almost 50% faster when processing the spatially organized quads in our captures.

3.5.1 Shading and Transparency. As our quad proxies may require correctly ordered alpha blending, we use a 3-stage process to reconstruct them on the client. First, we render all opaque quad proxies; next, we render all transparent quad proxies into a separate frame buffer. We then composite the two buffers together. The metadata of each quad proxy tells us whether it is opaque or transparent, allowing us to efficiently avoid processing unnecessary vertices and fragments. In the transparent rendering stage, we use atomics to capture the N closest fragments and their color. We use the depth map from the opaque pass for early Z rejection in the transparency pass; the stencil buffer from the transparent pass is used for masking fragment execution in the composition pass, and we launch work only where the transparency buffers are not empty. We note that fully resolving transparency is necessary only around object silhouettes and transparent objects. For non-transparent scenes, our quad merging strategy conveniently places small quads around the silhouettes of most objects (the exception being a flattened quad), so transparency resolution is executed only for the few pixels necessary around object boundaries.

REFERENCES

Pontus Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, Kalle Åström, and Mark D. Fairchild. 2020. FLIP: A Difference Evaluator for Alternating Images. *Proc. ACM Comput. Graph. Interact. Tech.* 3, 2 (2020), 15:1–15:23.